Programmation Tacticals

Florent Kirchner

National Institute of Aerospace
fkirchne@nianet.org

1 Tune-up

The first procedural theorem prover, Edinburg LCF [2], also introduced notion of *strategies* or *tacticals*, i.e., operators that combine the elementary proof tactics. Its descendants, PVS [3], Coq [1] and all the procedural theorem provers used their own version of the original set of strategies, mostly empirically build. In this paper we intend to expose how we extend the PVS language to a point where it can be viewed as a quite powerful programming language. The finality of this work is to provide a clear basis to allow users to efficiently program complex strategies.

Prerequisite: Ben DiVito's patterns for PVS [4].

1.1 Formalism

It seems interesting to be able to give reduction rules for each of the strategies we are exposing, thus implying the rapid presentation of a framework.

As in all reduction semantics, we need to define some reduction rules, a reduction context and some values. Here goes:

the reduction relation transforms a command e containing strategies and tacticals into a simpler command e', by reducting the head strategy. It writes:

$$e \;/\; \tau \xrightarrow{\epsilon} e' \;/\; \tau \;\;,$$

$\xrightarrow{\epsilon}$ denoting a head reduction, and τ being the proof context (see Appendix A.1 page 9 for detailed information about the proof context).

contexts define where reductions are allowed though the context rule:

$$\frac{e \;/\; \tau \xrightarrow{\epsilon} e' \;/\; \tau}{C[e] \;/\; \tau \longrightarrow C[e'] \;/\; \tau} \;.$$

In our case, we perform leftmost innermost reductions. Their detailed expression is given by the grammar exposed in Appendix A.2 page 10.

values noted v, they are the PVS's tactics, and two additional combinators $\vee_\tau^{m(x)}$ and $\wedge_\tau^{m(x)}$ whose description can be found in Appendix A.3 page 10.

2 Existing strategies

step ¶ Not explicitly an element of the syntax, ¶ marks the end of a command. It triggers the evaluation of the tactics and does the final parameter reset.
Reduction Rule:

$$v \ \P \ / \ \tau \xrightarrow{\epsilon} (v \ / \ \tau).\,\text{raisePointerToLeaf}().\,\text{setProgress(false)} \quad .$$

`(if` *lexpr step*1 *step*2`)` The lisp term *lexpr* is evaluated, if it reduces to `nil` then *step*2 is applied. Else *step*1 is applied.
Usage: `(if *-* (flatten -) (split +))` : if `*-*` (the list of formulas in the antecedent of the current goal) is not empty then apply disjunctive simplification to the antecedent, else apply conjunctive splitting to the consequent.
Reduction Rule:

$$\texttt{(if}\ t\ v_1\ v_2\texttt{)}\ / \ \tau \xrightarrow{\epsilon} \text{if } t = \texttt{nil} \text{ then } v_2 \ / \ \tau \text{ else } v_1 \ / \ \tau \quad .$$

`(let ((`x_1 *lexpr*$_1$`)...(`x_n *lexpr*$_n$`))` *step*`)` The local variable binding strategy. The symbols x_i are binded to the lisp expressions *lexpr*$_i$ in the latter bindings and in *step*.
Usage: `(let ((form-num car *+*)) (lift-if form-num))` fetches the first formula in the consequent of the current goal, and lifts the branching structure in this formula.
Reduction Rule:

$$\texttt{(let ((}x_1\ t_1\texttt{)...(}x_n\ t_n\texttt{))}\ e\texttt{)}\ / \ \tau \xrightarrow{\epsilon} e[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n] \ / \ \tau \quad .$$

`(try` *step*1 *step*2 *step*3`)` This strategy combines a branching facility triggered by the a progress condition, with an error catching functionnality. It applies *step*1 to the current goal, it this shows a progress then it applies *step*2, else it applies *step*3. Moreover, if *step*2 fails then this strategy returns *skip*.
Usage: `(try (flatten) (prop) (split))` applies `(flatten)` to the current goal; if it generates subgoals then the propositional simplification tactic is applied, else `(split)` is.
Reduction Rule:

$$\begin{aligned}(\texttt{try}\ v_1\ v_2\ v_3) \ / \ \tau \xrightarrow{\epsilon} \text{let } \chi = (v_1 \ / \ \tau) \text{ in} \\ \text{if } \chi.\,\text{hasProgressed}() \\ \text{then if } (v_2 \ / \ \chi) = \bot_n \text{ then } \texttt{(skip)} \ / \ \tau \\ \qquad \text{else } (v_1 \vee_\tau v_2) \ / \ \tau \\ \text{else } v_3 \ / \ \tau \quad .\end{aligned}$$

`(then &rest` *steps*`)` Applies all the *steps* in sequence, each to all the subgoals generated by the application of the previous.

Usage: `(then (flatten) (beta))` applies the disjunctive simplification to the current goal, and then performs β-reduction on all the generated subgoals.
Reduction Rule:

$$\texttt{(then}\ v_1 \ldots v_n\texttt{)}\ /\ \tau \xrightarrow{\epsilon} v_1 \vee_\tau \texttt{(then}\ v_2 \ldots v_n\texttt{)}\ /\ \tau\ .$$

`(then@ &rest` *steps*`)` Like `then` this strategy applies the *steps* in sequence, but each of them is applied to the first of the subgoals generated by the application of the previous.
Usage: `(then (flatten) (beta))` applies the disjunctive simplification to the current goal, and then performs β-reduction on the first of the generated subgoals.
Reduction Rule:

$$\texttt{(then}\ v_1 \ldots v_n\texttt{)}\ /\ \tau \xrightarrow{\epsilon} v_1 \vee_\tau^{\text{raisePointerToLeaf()}} \texttt{(then}\ v_2 \ldots v_n\texttt{)}\ /\ \tau\ .$$

`(repeat` *step*`)` *step* is applied to the current goal, if it generates any subgoal then it is recursively applied to the first of these subgoals. The repetition stops when an application of *step* has no effect.
Usage: `(repeat (do-rewrite))` repeatedly applies rewrite steps along the main proof branch until no progress results from it.
Reduction Rule:

$$\begin{aligned}\texttt{(repeat}\ v\texttt{)}\ /\ \tau \xrightarrow{\epsilon} &\text{if } (v\ /\ \tau).\,\text{hasProgressed}() \\ &\text{then } \left[v \vee_\tau^{\text{raisePointerToLeaf()}} \texttt{(repeat}\ v\texttt{)}\right]\ /\ \tau \\ &\text{else } \texttt{(skip)}\ /\ \tau\ .\end{aligned}$$

`(repeat*` *step*`)` Like `repeat`, this strategy repeats *step*, but on all the previously generated subgoals.
Usage: `(repeat* (do-rewrite))` repeatedly applies rewrite steps along all branches until no progress results from it.
Reduction Rule:

$$\begin{aligned}\texttt{(repeat*}\ v\texttt{)}\ /\ \tau \xrightarrow{\epsilon} &\text{if } (v\ /\ \tau).\,\text{hasProgressed}() \\ &\text{then } \left[v \vee_\tau \texttt{(repeat*}\ v\texttt{)}\right]\ /\ \tau \\ &\text{else } \texttt{(skip)}\ /\ \tau\ .\end{aligned}$$

`(spread` *step*0 `(`*step*1 $\ldots$ *step*N`))` First applies *step*0 and then each of the *step*I to one of the subgoals generated.
Usage: `(spread (flatten) ((ground) (assert) (lift-if)))` applies the disjunctive simplification step to the current goal, then apply `(ground)` to the first generated subgoal, `(assert)` to the second and `(lift-if)` to the third.
Reduction Rule:

$$\texttt{(spread}\ v_0\ \texttt{(}v_1 \ldots v_n\texttt{))}\ /\ \tau \xrightarrow{\epsilon}$$

$$
\begin{array}{l}
\text{let } \chi = (\bigwedge(v_0, \ldots, v_n) \mathbin{/} \tau) \text{ in} \\
\quad \text{if } \chi = \varnothing \text{ then } \texttt{(spread } v_0 \texttt{ (}v_1 \ldots v_{n-1}\texttt{))} \mathbin{/} \tau \\
\quad \text{else if } \chi.\,\text{pointNextSibling}() = \varnothing \\
\qquad \text{then } \left[\bigwedge(v_0, \ldots, v_n) \wedge_\tau^{\text{lowerPointer}(2)} \texttt{(skip)}\right] \mathbin{/} \tau \\
\qquad \text{else } \texttt{(spread } v_0 \texttt{ (}v_1 \ldots v_{n-1} v_n \texttt{(skip)))} \mathbin{/} \tau \ .
\end{array}
$$

`(branch` *step*0 `(`*step*1 $\ldots$ *step*N`))` This strategy behaves as `spread`, but if there are $M > N$ subgoals generated by *step*0 then it will apply *step*N to all the subgoals $N+1$, $N+2$, $\ldots M$.
Usage: `(branch (flatten) ((ground) (assert)))` applies the disjunctive simplification step to the current goal, then apply `(ground)` to the first generated subgoal, and `(assert)` to the rest of the subgoals.
Reduction Rule:

$$
\begin{array}{l}
\texttt{(branch } v_0 \texttt{ (}v_1 \ldots v_{n-1} v_n\texttt{))} \mathbin{/} \tau \xrightarrow{\epsilon} \\
\quad \text{let } \chi = (\bigwedge(v_0, \ldots, v_n) \mathbin{/} \tau) \text{ in} \\
\quad\quad \text{if } \chi = \varnothing \text{ then } \texttt{(branch } v_0 \texttt{ (}v_1 \ldots v_{n-1}\texttt{))} \mathbin{/} \tau \\
\quad\quad \text{else if } \chi.\,\text{pointNextSibling}() = \varnothing \\
\quad\quad\quad \text{then } \left[\bigwedge(v_0, \ldots, v_n) \wedge_\tau^{\text{lowerPointer}(2)} \texttt{(skip)}\right] \mathbin{/} \tau \\
\quad\quad\quad \text{else } \texttt{(branch } v_0 \texttt{ (}v_1 \ldots v_{n-1} v_n v_n\texttt{))} \mathbin{/} \tau \ .
\end{array}
$$

`(try-branch` *step*1 `(`*step*1$'$ $\ldots$ *step*N'`)` *step*2`)` A combination of the `try` and the `branch` strategies, `try-branch` applies *step*1 to the current goal, and in case it generated subgoals it applies each of the *step*I' to one of the subgoals. Else it applies *step*2. As for `try`, this strategy catches any failure that would arise from the application of any of the *step*I.
Usage: `(try-branch (flatten) ((ground) (assert)) (split))` applies `(flatten)` to the current goal. If it generated subgoals it applies `(ground)` to the first of these subgoals and `(assert)` to the rest of the subgoals. Else it applies `(split)`.
Reduction Rule:

$$
\texttt{(try-branch } v_1 \texttt{ (}v_1'\texttt{) } v_2\texttt{)} \mathbin{/} \tau \equiv \texttt{(try } v_1\ v_1'\ v_2\texttt{)} \mathbin{/} \tau \ .
$$

And:

$$
\begin{array}{l}
\texttt{(try-branch } v_1 \texttt{ (}v_1' \ldots v_n'\texttt{) } v_2\texttt{)} \mathbin{/} \tau \xrightarrow{\epsilon} \\
\quad \text{let } \chi = (v_1 \mathbin{/} \tau) \text{ in} \\
\quad\quad \text{if } \chi = \bot_n \text{ then } \texttt{(fail)} \mathbin{/} \tau \\
\quad\quad \text{else if } \chi.\,\text{hasProgressed}() \\
\quad\quad\quad \text{then if } (\texttt{(branch } v_1 \texttt{ (}v_1' \ldots v_n'\texttt{))} \mathbin{/} \tau) = \bot_n \\
\quad\quad\quad\quad \text{then } \texttt{(skip)} \mathbin{/} \tau \\
\quad\quad\quad\quad \text{else } \texttt{(branch } v_1 \texttt{ (}v_1' \ldots v_n'\texttt{))} \mathbin{/} \tau \\
\quad\quad\quad \text{else } v_2 \mathbin{/} \tau \ .
\end{array}
$$

3 New strategies

`(match` *pvsterm* `((`*pat*1 `->` *step*1`)...(`*patN* `->` *stepN*`)))` Matches *pvsterm* with each of the patterns *pat i*. On the first match found it returns the corresponding step, properly instanciated.
Usage: `(match (! -2 R) (("%1 / %2") -> (rewrite-lemma "ndiv_lt" -2 ("%1" "x" "%2" "b"))))` matches the right-hand side of formula -2 with a dividing pattern, if there is a match then it rewrites this formula with lemma `ndiv_lt` (stating that the result of the euclidian division of x by b is lower or equal to the corresponding real division).
Reduction Rule: Let $\oplus$ be the binary operator defined as:

$$\begin{aligned}
\sigma_1 e_1 \oplus \sigma_2 e_2 \;/\; \tau &\longrightarrow v_1 \;/\; \tau && \text{if the substitution } \sigma_1 \text{ is defined} \\
&&& \text{and } e_1 \;/\; \tau \text{ evaluates in } v_1; \\
&\longrightarrow v_2 \;/\; \tau && \text{else and if } \sigma_2 \text{ is defined} \\
&&& \text{and } e_2 \;/\; \tau \text{ evaluates in } v_2; \\
&\longrightarrow \text{Idtac} \;/\; \tau && \text{else.}
\end{aligned}$$

For all $i \in \{1, \ldots, n\}$, $\sigma_{p_i \leftarrow t}$ is the substitution resulting from the matching of t by p_i (undefined if p_i does not match t ; matching by _ always succeds and yields the empty substitution).
Then:

$$\texttt{(match } t \;\; (p_i \texttt{ ->} e_i)_{i=1}^{n}\texttt{)} \;\; / \, \tau \xrightarrow{\epsilon} \bigoplus_{i=1}^{n} \sigma_{p_i \leftarrow t} \, ei \;/\; \tau \;\; .$$

`(screen (...(`*apat*$_{i,1}$...*apat*$_{1,n_i}$ `\-` *cpat*$_{i,1}$...*cpat*$_{1,n_i}$ `->` *step*$_i$`)...))` This strategy matches the proof context against a set of patterns. The order of the patterns is not decisive; `#`n designates the formula that was matched by the nth pattern. When a match is found, it applies the corresponding step, properly instanciated. If the \− symbol is omited, the pattern will be checked against consequent and antecedent formulas.
Usage: `(screen (("%1 * %1 = 0" \- "%2 > %3" -> (rewrite "sqrt_1" "#1")) ("" -> (grind))))` matches the current goal; if the antecedent contains a formula with a squared term equated to 0 and the consequent contains a formula with a "greater than" symbol, then it rewrites the formula matched by the first pattern with a lemma concerning null squares. If no match is found satisfying these conditions, it applies `(grind)`.
Reduction Rule: Here a simplified rule is presented, the complete rule being a simple (but space consuming) extension.

$$\texttt{(screen } (hp_i \vdash p_i \texttt{ -|} e_i)_{i=1}^{n}\texttt{)} \;\; / \, \tau . (\ldots A_j \ldots \vdash B) \xrightarrow{\epsilon} \bigoplus_{i=1}^{n} \sigma_{hp_i \leftarrow A_j} \sigma'_{p_i \leftarrow B} \, e_i \;/\; \tau \;\; .$$

If this does not succeed then the context progression rule is used instead:

$$\texttt{(screen } (hp_i \vdash p_i \texttt{ -|} e_i)_{i=1}^{n}\texttt{)} \;\; / \, \tau . (\ldots A_j \ldots \vdash B) \xrightarrow{\epsilon}$$

$$\texttt{(screen}\ (hp_i \vdash p_i \twoheadrightarrow e_i)_{i=1}^{n}\texttt{)}\ /\ \tau.(\ldots A_{j-1}\ldots \vdash B)\ .$$

`(throw` *tag*`)` fails with the name *tag*.
Usage: `(throw "fatalError")` throws an error with the name "fatalError".
Reduction Rule:

$$\texttt{(throw}\ t\texttt{)}\ /\ \tau \hookrightarrow \bot_t$$

`(catch` *step*1 `&optional` *tag step*2`)` If *step*1 throws an error whose name corresponds to *tag* then run *step*2. If no tag is provided or if the tag is the empty string then the strategy catches any error that *step*1 creates. If no *step*2 is provided then it is assumed to be a (*skip*).
Usage: `(catch (case "x > 0") "" (split))` applies a case analysis on the current sequent, on the variable `x`. If this fails then it applies the conjunctive splitting rule.
Reduction Rule:

$$\begin{aligned}\texttt{(catch}\ v_1\ t\ v_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} &\text{if } (v_1\ /\ \tau) = \bot_t \text{ then } v_2\ /\ \tau\\ &\text{else if } (v_1\ /\ \tau) = \bot_u \text{ then } \texttt{(throw}\ u\texttt{)}\ /\ \tau\\ &\quad\text{else } v_1\ /\ \tau\ .\end{aligned}$$

`(try!` *step*1 *step*2 *step*3`)` This is a "strict" try, that does not do error handling. If *step*1 generates subgoals, then *step*2 is applied to all of them, else *step*3 is applied.
Usage: `(try! (flatten) (propax) (split))` applies `(flatten)` to the current goal; if it generates subgoals then the `(propax)` tactic is applied, else `(split)` is. If any of these tactics fail, it fails.
Reduction Rule:

$$\begin{aligned}\texttt{(try!}\ v_1\ v_2\ v_3\texttt{)}\ /\ \tau \xrightarrow{\epsilon} &\text{let } \chi = (v_1\ /\ \tau) \text{ in}\\ &\text{if } \chi.\,\text{hasProgressed}()\\ &\text{then } (v_1 \vee_\tau v_2)\ /\ \tau\\ &\text{else } v_3\ /\ \tau\ .\end{aligned}$$

`(when` *lterm step*1 `&rest` *step*2`...`*stepN*`)` If the lisp term *lterm* evaluates in `nil` then this strategy returns (*skip*). Else it applies *step*1`...` *stepN* in a sequence alobg the proof's main branch.
Usage: `(when (done-subgoals *ps*) (skolem!))` applies variable skolemization to the current goal if there are no proved subgoals in the proof state.
Reduction Rule:

$$\begin{aligned}\texttt{(when}\ t\ v_1\ldots v_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} &\text{if } t = \texttt{nil}\ \text{ then } (\mathit{skip})\ /\ \tau\\ &\text{else } \texttt{(then@}\ v_1\ldots v_n\texttt{)}\ /\ \tau\ .\end{aligned}$$

`(when*` *lterm step*1 `&rest` *step*2...*stepN*`)` If the lisp term *lterm* evaluates in `nil` then this strategy returns (*skip*). Else it applies *step*1... *stepN* in an all-branches sequence.
Usage: `(when* (done-subgoals *ps*) (skolem!))` applies variable skolemization to the current goal if there are no proved subgoals in the proof state.
Reduction Rule:

$$\texttt{(when*}\ t\ v_1 \ldots v_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} \text{if } t = \texttt{nil}\ \text{ then } (\textit{skip})\ /\ \tau \quad \text{else } \texttt{(then}\ v_1 \ldots v_n\texttt{)}\ /\ \tau\ .$$

`(while` *lterm step*1 `&rest` *step*2...*stepN*`)` When the lisp term *lterm* is non-`nil`, this strategy applies repeatedly *step*1... *stepN* along the main branch of the proof.
Usage: `(while (pending-subgoals *ps*) (skolem!))` applies variable skolemization to the current goal, and then to each of the first subgoal generated, as long as there are unproved subgoals in the proof state and that the proof progresses.
Reduction Rule:

$$\texttt{(while}\ t\ v_1 \ldots v_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} \texttt{(repeat (when}\ t\ v_1 \ldots v_n\texttt{))}\ /\ \tau\ .$$

`(while*` *lterm step*1 `&rest` *step*2...*stepN*`)` This strategy behaves just as `while`, but eventually repeats *step*1... *stepN* on all the branches of the proof.
Usage: `(while* (pending-subgoals *ps*) (skolem!))` applies variable skolemization to the current goal, and then simultaneously to all of the subgoals generated, as long as there are unproved subgoals in the proof state and that the proof progresses.
Reduction Rule:

$$\texttt{(while}\ t\ v_1 \ldots v_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} \texttt{(repeat* (when}\ t\ v_1 \ldots v_n\texttt{))}\ /\ \tau\ .$$

`(for` *lint step*`)` Here *lint* is a lisp integer. This strategy repeats *step*, *lint* times, along the main branch of the proof. If *lint* is negative, the strategy is equivalent to the `(repeat` *step*`)` strategy.
Usage: `(for 3 (beta))` applies the β-reduction rule three times, first on the current goal and then on the first of the generated subgoals.
Reduction Rule

$$\texttt{(for}\ n\ v\texttt{)}\ /\ \tau \xrightarrow{\epsilon} \text{if } n = \texttt{0}\ \text{ then } (\textit{skip})\ /\ \tau \quad \text{else if } n < \texttt{0}\ \text{ then } \texttt{(repeat}\ \textit{step}\texttt{)}\ /\ \tau \quad \text{else } \texttt{(then@}\ v\ \texttt{(for@}\ (n-1)\ v\texttt{))}\ /\ \tau\ .$$

`(for*` *lint step*`)` Behaves as the `for` strategy, but applies *step* on all the branches of the proof.
Usage: `(for* 3 (beta))` applies the β-reduction rule three times, first on the current goal and then on all of the generated subgoals.

Reduction Rule

$$\texttt{(for*}\ n\ v\texttt{)}\ /\ \tau \xrightarrow{\epsilon} \text{if } n = \texttt{0}\ \text{ then } (\mathit{skip})\ /\ \tau \\ \text{else if } n < \texttt{0}\ \text{ then } \texttt{(repeat*}\ \mathit{step}\texttt{)}\ /\ \tau \\ \text{else } \texttt{(then}\ v\ \texttt{(for@}\ (n-1)\ v\texttt{))}\ /\ \tau\ .$$

`(first (`*step*1`...`*stepN*`))` Applies the first of the steps that does not fail. If no step fullfill such a condition, the strategy fails.
Usage: `(first (case "y > 0") (bddsimp) (skip))` tries to apply the case analysis command to the current goal, if it fails it tries the propositional simplification. If this also fails, it applies the `(skip)` tactic (which cannot fail).
Reduction Rule: We give a recursive definition of the reduction rule:

$$\texttt{(first ())}\ /\ \tau \xrightarrow{\epsilon} \texttt{(fail)}\ /\ \tau\ .$$

$$\texttt{(first (}v_1 \ldots v_n\texttt{))}\ /\ \tau \xrightarrow{\epsilon} \text{if } (v_1\ /\ \tau) \neq \bot \text{ then } v_1\ /\ \tau \\ \text{else } \texttt{(first (}v_2 \ldots v_n\texttt{))}\ /\ \tau\ .$$

`(solve (`*step*1`...`*stepN*`))` This strategy selects and applies the first of its arguments that will prove the current goal. If it has no such argument, it fails.

Usage: `(first (case "y > 0") (bddsimp))` tries to apply the case analysis command to the current goal, if it does not completely prove the current goal it tries the propositional simplification. If this also fails to completely prove the current goal, it fails.
Reduction Rule: We give a recursive definition of the reduction rule:

$$\texttt{(solve ())}\ /\ \tau \xrightarrow{\epsilon} \texttt{(fail)}\ /\ \tau\ .$$

$$\texttt{(solve (}v_1 \ldots v_n\texttt{))}\ /\ \tau \xrightarrow{\epsilon} \text{if } (v_1\ /\ \tau).\text{isActiveTreeProved}() \text{ then } v_1\ /\ \tau \\ \text{else } \texttt{(solve (}v_2 \ldots v_n\texttt{))}\ /\ \tau\ .$$

`(piks)` is not a strategy as we defined them since it does not act as a combinator of tactics, but more as a tactic. It does not do anything special, but does not either trigger the `"No change on..."` reaction of PVS. Basically, `(piks)` is used in strategy writing to deceive the progress-testing strategies.

Usage: `(try (piks) (flatten-disjunct) (skip))` is a simple way to catch any error generated by the application of the controlled disjunctive simplification rule.
Reduction Rule:

$$\texttt{(piks)}\ /\ \tau \hookrightarrow \tau.\,\text{setProgress(true)}$$

4 Conclusion

This work provides some powerful strategies to enables efficient strategy creation.

A Framework and notations

A.1 The proof context

First, a sequent is represented as $\Gamma \vdash \Delta$, where Γ is the *antecedent* and Δ is the *consequent*, each being a list of formulas[1]. Latin letters A, B, etc. represent individual formulas.

The proof context is considered as a collection of sequents organized in a tree of sequents, its leaves representing the sequents that are currently being proved. A leaf, when modified by some command, becomes the father of the sequents created by this command: the nodes of the tree of sequents are the "old" sequents. Thus, the tree of sequents keeps trace of the proof progression. Incidentally, one has to consider the number of features that are relied to the proof context (state of the proof, proved branches, goal numbering, etc.) Hence we blend a simplified object-oriented structure with the tree representation. We write $O.m(\bar{x})$ for the invocation of the method m of object O with the list of parameters $\bar{x}$. Methods modifying an object return a new object, whereas methods that test properties leave the object unchanged. Thus, a method call $O.m(\bar{x})$ is a synonymous for the function call $m(\bar{x}, O)$, and the objects could also be seen as records. The letter τ denotes a proof context object; we distinguish a few particular proof contexts:

- $\top$ is a proof context that is completely proved.
- $\bot_s$ stands for a failed proof context (the string s codes for an error tag).
- And $\varnothing$ is the empty proof context, i.e., a proof context object hosting an empty tree.

Follows the description of the attributes and methods of τ.

- Attributes:
  - $\tau.\mathrm{seq\_tree}$: the tree of sequents.
  - $\tau.\mathrm{active}$: pointer to the active subtree of sequents, i.e., the subtree on which the next command will take effect. In case it is a leaf, then $\tau.\mathrm{active}$ represents a sequent $\Gamma \vdash \Delta$, and we will write $\tau.\ \Gamma \vdash \Delta$.
  - $\tau.\mathrm{progress}$: this is a flag raised when the tree of sequents has gone through changes.
- Methods:
  - $\tau.\mathrm{addLeafs}(\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n)$: this method adds n leafs to the tree. In the new tree, the new sequents $\Gamma_i \vdash \Delta_i$, $i \in \{1, \ldots, n\}$, will be leaves, and the former active leaf of the old tree will become their common node.
  - $\tau.\mathrm{lowerPointer}(i)$: moves the active pointer down (towards the root) in the tree, i being the depth of the move.
  - $\tau.\mathrm{raisePointerToLeaf}()$: moves the pointer up to the first unproved leaf of the tree.

[1] The semantics presented in this paper does not distinguish between sequents with permuted formulas. This limitation is not problematic since we focus on tacticals, which do not require formula-level knowledge.

- τ.pointNextSibling(): moves the pointer to the closest unproved leaf, sibling of the active sequent. If there is no such sibling, the pointer is set to a default empty value, which is represented by the method returning the empty proof context $\varnothing$.
- τ.setProgress(b): sets the corresponding flag to b.
- τ.hasProgressed(): returns the value of the progress flag.
- τ.setLeafProved(): the active leafs are labeled as proved. If there are no unproved sequents left, the proof is finished (i.e., τ.setLeafProved() $= \top$).
- τ.isActiveTreeProved(): returns true if all the leafs in the active subtree are labeled as proved, false otherwise.

A.2 The context grammar

$$
\begin{aligned}
C ::= \; & [\,] \\
& \mid C \vee^{\mathrm{f}}_{\tau} e \mid v \vee^{\mathrm{f}}_{\tau} C \\
& \mid C \wedge^{\mathrm{f}}_{\tau} e \mid v \wedge^{\mathrm{f}}_{\tau} C \\
& \mid C \P \\
& \mid \texttt{(if } t\ C\ e_1\texttt{)} \\
& \mid \texttt{(if } t\ e_1\ C\texttt{)} \\
& \mid \texttt{(try } C\ e_2\ e_3\texttt{)} \\
& \mid \texttt{(try } v_1\ C\ v_3\texttt{)} \\
& \mid \texttt{(try } v_1\ v_2\ C\texttt{)} \\
& \mid \texttt{(repeat } C\texttt{)} \\
& \mid \texttt{(repeat* } C\texttt{)} \\
& \mid \texttt{(spread } C\ (e'_1 \ldots e'_n)\texttt{)} \\
& \mid \texttt{(spread } v_1\ (C \ldots e'_n)\texttt{)} \\
& \mid \ldots \mid \texttt{(spread } v_1\ (v'_1 \ldots C)\texttt{)} \\
& \mid \texttt{(try-branch } C\ (e'_1 \ldots e'_n)\ e_2\texttt{)} \\
& \mid \texttt{(try-branch } v_1\ (C \ldots e'_n)\ e_2\texttt{)} \\
& \mid \ldots \mid \texttt{(try-branch } v_1\ (v'_1 \ldots C)\ e_2\texttt{)} \\
& \mid \texttt{(try-branch } v_1\ (v'_1 \ldots v'_n)\ C\texttt{)} \;.
\end{aligned}
$$

A.3 The $\wedge_{\tau}^{m(x)}$ and $\vee_{\tau}^{m(x)}$ special tactics

We define two operators to express some fundamental operations:

$$
\begin{aligned}
\vee_{\tau}^{m(\bar{x})} &: \mathit{tactic} \times \mathit{tactic} \longrightarrow \mathit{tactic} \\
\wedge_{\tau}^{m(\bar{x})} &: \mathit{tactic} \times \mathit{tactic} \longrightarrow \mathit{tactic} \;.
\end{aligned}
$$

The first operator $\vee_{\tau}^{m(\bar{x})}$, applies two tactics and the method $m(\bar{x})$ in sequence, unless the first tactic has failed or proved the subtree. The second operator, $\wedge_{\tau}^{m(\bar{x})}$, is also a sequential command, but only stops if the proof has failed. Both these operators will be added as values of our semantics. Using these two operators, one can build a macro $\bigwedge$ that applies the first element of a list of tactics to a goal, and each of the next elements to a subgoal:

$$\bigwedge : tactic^n \longrightarrow tactic \ ,$$

recursively defined as:

$$\begin{aligned} \bigwedge(p_0, \ldots, p_n) &\longrightarrow \bigwedge(p_1, \ldots, p_{n-1}) \wedge_{\tau}^{\text{pointNextSibling()}} \ p_n \\ \bigwedge(p_0, p_1) &\longrightarrow p_0 \vee_{\tau}^{\text{raisePointerToLeafs()}} \ p_1 \ . \end{aligned}$$

Using the definition of tactics reductions:

$$p \ / \ \tau \hookrightarrow \tau' \ ,$$

we now can give a formal definition of the aforementioned operators:

$$\begin{aligned} p_1 \vee_{\tau}^{m(\bar{x})} p_2 \ / \ \tau \hookrightarrow \text{let } \chi = p_1 \ / \ \tau \text{ in} \\ \text{if } \chi.\text{isActiveTreeProved() then } \chi \\ \text{else if } (\chi = \bot_n) \text{ then } \chi \\ \text{else } p_2 \ / \ \chi.m(\bar{x}) \ , \end{aligned}$$

and:

$$\begin{aligned} p_1 \wedge_{\tau}^{m(\bar{x})} p_2 \ / \ \tau \hookrightarrow \text{let } \chi = p_1 \ / \ \tau \text{ in} \\ \text{if } (\chi = \bot_n) \text{ then } p_1 \\ \text{else } p_2 \ / \ \chi.m(\bar{x}) \ . \end{aligned}$$

When no method is provided for the two operators, m is assumed to be the identity method.

References

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[2] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *lncs*. sv, 1979.

[3] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[4] Ben L. Di Vito. A PVS prover strategy package for common manipulations. Technical Report TM-2002-211647, Langley Research Center, Hampton, VA, April 2002.

Index